NP-completeness

Reading

Section 7.4

Practice problems (page 294):

NP-complete problems

NP-complete problems is a class of problems that can be considered to be the "hardest" NP problems. The discovery of NP-complete problems was a major development for the field.

But first, an appropriate definition of reducibility is in order:

> We say that the language A is **polynomial time mapping reducible** to a language B, and written
>
> $$A \leq_P B$$
>
> , if a polynomial time computable function
>
> $$f : \Sigma^* \to \Sigma^*$$
>
> exists that is a mapping reduction, namely:
>
> $$w \in A \text{ if and only if } f(w) \in B$$
>
> We say that the function is polynomial time computable, if there is a polynomial time Turing Machine that on input w halts leaving $f(w)$ on the tape.

This is nothing more than the notion of mapping reducibility adjusted to take into account polynomial running time concerns. The following theorem should be easy to argue:

> If $A \leq_P B$ and $B \in P$, then $A \in P$.

Let us show this, by writing the Turing Machine for P:

- Start with input w of size n.
- Use the TM that implements the mapping reduction, to have $f(w)$ remain at the tape.
- Since this algorithm runs in time polynomial to its input, the size m of $f(w)$ must be a polynomial in n.
- We then run the polynomial time TM that recognizes the language B, and accept if it accepts, reject if it rejects.
- This TM takes time polynomial in the size m of its input $f(w)$. This in turn is polynomial in n.
- Adding the two running times remains polynomial in n.

The Satisfiability problem

A language particulary important to the discussion of NP-completeness is known as SAT:

> SAT is the language of all string representations of satisfiable Boolean formulas.
>
> A **boolean formula** is built out of boolean variables x, y etc that take values 1 for true and 0 for false, and have the standard boolean operations AND ($x \wedge y$), OR ($x \vee y$) and complement $\bar{x}$.
>
> A boolean formula is **satisfiable**, if there are assignments to the variables that make the formula evaluate to 1.

In essence, the SAT problem asks whether there is an algorithm/Turing Machine that given a boolean formula can decide if it is satisfiable or not. The main result, known as the Cook-Levin theorem, says:

> **Theorem (Cook-Levin)**
>
> $$\text{SAT} \in P \text{ if and only if } P = NP$$

In a certain sense then, SAT is the "hardest possible NP problem": If *it* is solvable in polynomial time, then all NP problems are solvable in polynomial time.

A related problem is 3SAT, which deals only about formulas that are in **3-conjunctive normal form**. These are the conjunctions (ANDs) of disjunctions (ORs) of 3 variables or their complements. So a 3cnf formula would look like this:

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee x_1 \vee \bar{x}_1)$$

To satisfy such a formula, you need to simultaneously satisfy each of the 3-term disjunctive clauses.

The following should be clear:

$$\text{3SAT} \leq_P \text{SAT}$$

It should be also clear that SAT and 3SAT are both in NP. A non-deterministic time machine for them simply correctly guesses variable assignments for them, then verifies those variable assignments work.

More interesting is the fact that we can relate these satisfiability problems to other problems like CLIQUE:

3SAT and CLIQUE

3SAT is polynomial time reducible to CLIQUE

In order to answer this question we need to do the following:

- We start with a formula $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) land \cdots \wedge (a_k \vee b_k \vee c_k)$.
- We need from that formula to produce, in polynomial time, an instance of CLIQUE, which consists of a graph G and an integer k. It turns out that k is actually the number of clauses in ϕ.
- We show that ϕ is satisfiable if and only if the corresponding graph has a k-clique.

The graph is created as follows:

- We create $3k$ vertices in G, organized in groups of 3.
- Each group corresponds to one of the clauses, containing one node for each of the 3 terms in the clause. These nodes/vertices are "labeled" by the terms.
- We now add edges as follows:
  - No edges between nodes in the same 3-group. This ensures that a k-clique contains exactly one node from each 3-group.
  - Two other nodes are connected if their terms can be satisfied simultaneously. The only time that does not happen is if the one term is a variable x and the other term is its complement $\bar{x}$.
  - In other words if two nodes from different 3-groups are not connected, that means that we cannot satisfy those terms simultaneously.

Now we have to show this is a reduction:

- Suppose ϕ is an actual 3SAT instance
  - meaning we can assign values to the variables so that the whole formula is satisfied.
  - Then each of the 3-clauses must evaluate to 1 (true).
  - This means that on each clause there must be at least one term that is 1.
  - If we look at the corresponding vertices in G, then they will form a k-clique, as there are exactly k clauses altogether. It is a clique, because if both terms are 1 they are going to be connected, as they cannot be a variable and its negation.
- Conversely, if the graph has a k-clique
  - then that clique must consist of one term from each 3-group, as terms in the same 3-group are not connected so we must use only one of each.
  - The fact those terms are all connected means that we can assign values to the variables in those terms so that they all equal 1.
  - That makes each 3-clause true, hence makes the whole formula ϕ true.

- We can assign arbitrary values to any remaining variables.
- We therefore see that ϕ is satisfiable.

A consequence of this is that if CLIQUE was in P, then 3SAT would also be in P.

We can similarly link many problems.

NP-completeness

We now define the notion of NP-completeness.

> A language B is **NP-complete** if:
>
> 1. B is in NP.
> 2. For any A in NP we have $A \leq_P B$.

In other words, NP-complete problems are such that all other NP problems are polynomial-time reducible to them. So if any NP-complete problem was proven to be in P, we would have that $P = NP$.

It would seem that to show a problem is NP-complete is hard. It turns out that only the first one is hard:

> If B is NP-complete and $B \leq_P C$ for some C in NP, then C is also NP-complete.
>
> So to show a problem C is NP-complete we have to:
>
> 1. Show it is in NP
> 2. Show that there is a polynomial time reduction from some NP-complete problem to C.

This makes it easy to prove problems are NP-complete. For example, after we show that 3SAT is NP-complete, the work we did in the previous section would show that CLIQUE is also NP-complete. This has led to the creation of a rich set of NP-complete problems.

Let us prove this fact now.

- We start with an NP-complete language B and an NP language C, and let $B \leq_P C$.
- We just need to prove the second condition, namely that for any other language A in NP we have $A \leq_P C$.
  - If A is in NP and since B is NP-complete, we have $A \leq_P B$.
  - Since also $B \leq_P C$, then we obtain $A \leq_P C$.

Cook-Levin Theorem

The Cook-Levin theorem shows that there is a in fact an NP-complete language, namely SAT. We will see a little while later that 3SAT is also NP-complete.

> **Theorem (Cook-Levin)**
>
> SAT and 3SAT are NP-complete.

We look at SAT for now. We have already seen that it is in NP.

The book has a very detailed proof of this theorem. The idea is based on considering a non-deterministic Turing machine that decides a language $A \in NP$ in n^k time, and builds a formula that captures the computation, using the idea of a tableau. The details, which you should study, are on pages 277 through 282.